COVER SHEET

A Move Generating Algorithm for Hex Solvers

Rune Rasmussen, Frederic Maire and Ross Hayward

Faculty of Information Technology,
Queensland University of Technology,
Gardens Point Campus,
GPO Box 2434,
Brisbane QLD 4001,
Australia.
r.rasmussen@qut.edu.au,
f.maire@qut.edu.au,
r.hayward@qut.edu.au

Abstract. [1]Generating good move orderings when searching for solutions to games can greatly increase the efficiency of game solving searches. This paper proposes a move generating algorithm for the board game called *Hex*, which in contrast to many other approaches, determines move orderings from knowledge gained during the search. This *move generator* has been used in Hex searches solving the 6x6 Hex board with comparative results indicating a significant improvement in performance. One anticipates this move generator will be advantageous in searches for complete solutions of Hex boards, equal to, and larger than, the 7x7 Hex board.

1 Introduction

Programs that can solve two player games explore a hierarchy of board positions. This hierarchy, known as a *game tree*, is a rooted tree whose nodes are all valid board positions and whose edges are legal moves [1]. Programs that analyze or solve games, such as Chess, Go or Hex, use minimax search algorithms with pruning to search their respective game trees [2, 3][4][5]. Pruning is maximized if moves that trigger the pruning are searched first. The benefit of good move orderings is a smaller search. Move generating algorithms are known as *move generators* and they place moves in an order which maximize pruning. Move generators which generate moves as a function of a single board position are called *static*. A *dynamic* move generator generates moves as a function of a game tree [6]. Such move generators sample some or all of the board positions in a game tree to evaluate moves at the root board position.

This paper presents a dynamic move generator applied to the problem of solving the game of Hex. We demonstrate this move generator in a minimax search which van Rijswijck defines as the *Pattern Search* algorithm [7]. Hayward et al. apply

[1] To appear in: Lecture Notes in Artificial Intelligence, Springer (2006).

a refinement of the pattern search algorithm in their *Solver* program, which can solve small Hex boards up to and including the 7x7 Hex board [2, 3]. In Section 2, we introduce the game of Hex. Section 3, gives an overview of the pattern search algorithm. In Section 4, we describe the details of our move generator. Section 5 gives experimental results for our move generator in solving small Hex boards.

2 The Game of Hex

The game of Hex is a two-player game played on a tessellation of hexagonal cells which covers a rhombic board (see Figure 1)[8]. Each player has a cache of coloured stones. The goal for the player with the black stones is to connect the black sides of the board. Similarly, the goal for the player with the white stones is to connect the white sides of the board. The initial board position is empty. Players take turns and place a single stone, from their respective cache, on an empty cell. The first player to connect their sides of the board with an unbroken chain of their stones is the winner. The game of Hex never ends in a draw [8].

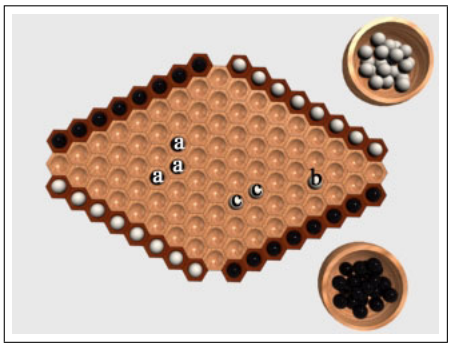

Fig. 1. A 9x9 Hex board.

If a player forms an unbroken chain of stones, not necessarily between that player's sides, then any two stones in that chain are said to be *connected*. A *group* is a maximal connected component of stones [9, 10]. Figure 1 shows seven groups where three of the seven groups have the labels a, b and c, respectively. The four sides of the board also constitute four distinct groups. A player wins a game, when the opposite sides for that player connect.

For the analysis of board positions, Anshelevich introduces the concept of a *sub-game* [9, 10]. In a sub-game, the players are called *Cut* and *Connect*. Both *Cut* and *Connect* play on a subset of the empty cells between two disjoint targets,

where a *target* is either an empty cell or one of *Connect's* groups. The player's roles are not symmetric, as *Connect* moves to form a chain of stones connecting the two targets, while *Cut* moves to prevent *Connect* from forming any such chain of stones. In this paper, we generalize sub-games to also include a subset of *Connect's* stones.

Definition 1 (sub-game). *A sub-game is a four-tuple* (x, S, C, y) *where* x *and* y *are targets. The set* S*, is a set of cells with Connect's stones and the set* C *is a set of empty cells. Finally,* x*,* y*,* S *and* C *are all disjoint.*

The set S is called the *support* and the set C is called the *carrier*. Anshelevich's sub-game is the case where S is the empty set [9, 10]. A sub-game is a *virtual connection* if *Connect* can win this sub-game against a perfect *Cut* player. A virtual connection is *weak* if *Connect* must play first to win the sub-game. In addition, a virtual connection is *strong* if *Connect* can play second and still win the sub-game. Yang et al. define a *threat pattern* as a virtual connection, where the targets are two opposite sides of the board [11]. There are a number of rules to deduce virtual connections (see [12]), however, the most relevant deduction rule for this paper is the OR deduction rule [9, 10].

Theorem 1 (OR Deduction Rule). *Let* (x, S, C_i, y) *be a set of* n *weak virtual connections with common targets* x *and* y *and common support* S*. If* $\bigcap_{i=0}^{n} C_i = \emptyset$*, then the sub-game* $(x, S, \bigcup_{i=0}^{n} C_i, y)$ *is a strong virtual connection.*

Let $M = \bigcap_{i=0}^{n} C_i$. If $M \neq \emptyset$ then *Cut* must move on a cell in M, otherwise, *Connect* has a move which can form a strong virtual connection. Hayward et al. call the set M, the *must-play region* [2, 3].

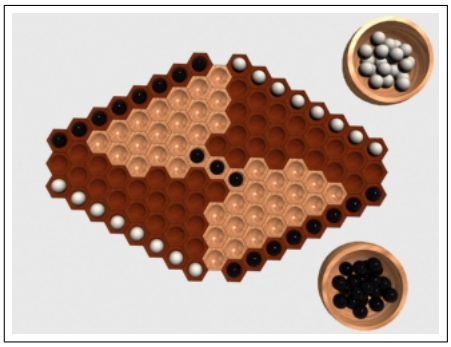

Fig. 2. An example of a threat pattern for player Black.

This paper will restrict its discussion to threat patterns. Figure 2, gives a view of a threat pattern where Black is *Connect*. The lightly colored empty cells form the carrier and the black stones at the centre of the board form the support.

3 Pattern Search

The *Pattern Search* algorithm, by van Rijswijck, is a game tree search which deduces threat patterns [7]. Hayward et al. apply the *Pattern Search* algorithm in their Solver program to solve small Hex boards [2, 3]. The pattern search algorithm is a depth-first traversal of the game tree, which deduces threat patterns as it backtracks from terminal board positions. The search switches between two modes, *Black mode* or *White mode*. In Black mode, the search tries to prove threat patterns for player Black and in White mode the search tries to prove threat patterns for player White.

Figure 3 gives an example of a search on a 3x3 Hex board in White mode. Player White is *Connect* and player Black is *Cut*. The diagram displays the carrier and the support on lightly coloured cells. The numbers on the stones indicate the order of the player's moves. On the left branch, the search visits terminal board position D where *Connect* is the winner. As the search backtracks from this terminal board position it removes *Connect's* move. At board position B, *Connect* has a weak threat pattern. The search backtracks once more and removes *Cut's* move. The search does a similar traversal of the right branch and deduces a weak threat pattern at board position C. At board position A, the search applies the OR deduction rule on the weak threat patterns found at B and C to deduce a strong threat pattern. On deducing that strong threat pattern, the search can backtrack and deal with board position A as it did with terminal board positions D and E.

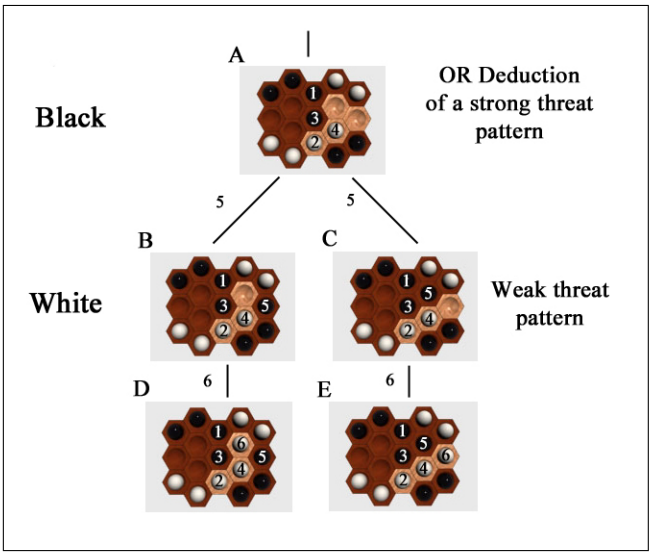

Fig. 3. The pattern search process for constructing threat patterns.

A cut-off condition and a switch of mode colour can occur if Black has a winning move elsewhere in this search. Figure 4 follows on from the previous pattern search example where the search is in White mode. The search backtracks from board position A, where it removes *Connect's* move. Board position X has a weak threat pattern. The search backtracks to board position Z and removes Black's move. At Z, the search tries another move which is a winning move for player Black. This winning move results from the search deducing a strong threat pattern for Black at board position Y. The existence of a strong threat pattern for Black at Y indicates a search cut-off condition and a switch of mode colour. The search abandons White's threat pattern at Z and switches to Black mode. The search deals with Y as it did with terminal board positions D and E. The switch of mode colour means player Black is now *Connect* and player White is now *Cut*.

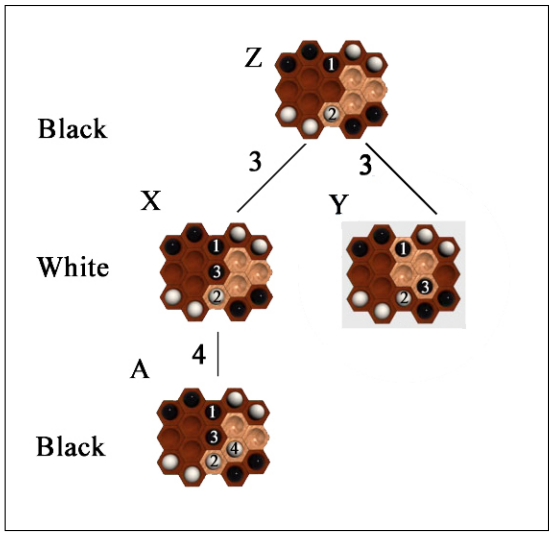

Fig. 4. The search is in White mode and switches to Black mode because there is a strong threat pattern for Black at Y.

4 A Dynamic Move Generator for Hex Solvers

Given the example of Figure 4, the aim of our move generator at board position Z is to derive, from the threat patterns found in the subtree under board position X, a good move order. Since Black is the player who moves at board position Z and Black is *Cut*, the problem for our move generator is to order moves for player *Cut*. Given board position Z, our move generator recommends a move on

the cell which has appeared most often with a *Connect* stone on terminal board positions where *Connect* is the winner.

Left of Figure 5 shows board position Z (from Figure 4) with *Connect's* weak threat pattern. Each empty cell in the carrier is labeled with the number of terminal board positions where that cell had player *Connect's* stone and *Connect* was the winner. Right of Figure 5 gives board position Y (also from Figure 4), where *Cut* has moved on that empty cell with the largest number, successfully cutting *Connect's* weak threat pattern.

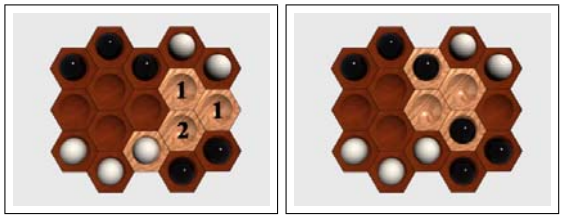

Fig. 5. Left: Each carrier cell has the number of times White moved on that cell to connect the targets. Right: Black's cutting move is the move White used most often to connect.

In the search for a weak threat pattern, the *Connection Utility* of a cell in the carrier of that weak threat pattern, is the number of terminal board positions where that cell has *Connect's* stone and *Connect* is the winner. Our move generator assumes that the goodness of a move for *Cut* on a cell is related to the connection utility of that cell. Our move generator is based on the following hypothesis.

Hypothesis 1. *The higher the connection utility the better is the move for Cut.*

When a succession of losing moves is generated for a given board position, our search derives from them a collection of weak threat patterns. Our move generator must derive the connection utilities for cells in the must-play region of these weak threat patterns. Figure 6 shows the derivation of connection utilities in a must-play region. The search is in White mode. The number on each carrier cell is its connection utility. The connection utilities on cells at board position A is the sum of connection utilities found at both board positions B and C and is restricted to the must-play region. From board position A, the search makes a move for Black on the cell with the largest connection utility. Board position D gives Black's winning threat pattern which causes a switch from White mode to Black mode. Board position D is treated as a terminal board position.

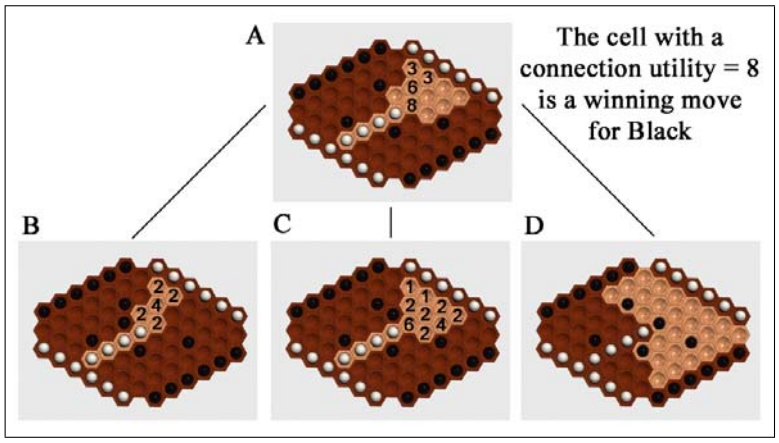

Fig. 6. The accumulation of connection utilities in a must-play region reveals a good move for Black.

Algorithm 1, the *Connection Utility Search* algorithm (see Appendix), is an extension of van Rijswijck's pattern search algorithm with our move generator approach. Lines 6, 7 and 8 initializes connection utility vectors from terminal board positions. For every cell with a *Connect* stone on a given terminal board position, the value one is assigned to the corresponding element in a vector of connection utilities. Line 30 is a vector sum of connection utilities. This sum occurs for each new weak threat pattern found at a given board position. Line 32 reorders the moves according to the new vector of connection utilities. Moves are restricted to the must-play region at line 31. Algorithm 1 is a recursive procedure which takes an initial board position, performs a search and returns a carrier, a vector of connection utilities and a mode colour. The returning mode colour is the colour of the winning player. If the winning player is the first player then the returning carrier belongs to a weak threat pattern, otherwise, it belongs to a strong threat pattern. Algorithm 1 works explicitly on the carriers of threat patterns because their respective targets and support are implicit in the search. In execution, the mode colour changes if the mode colour prior to line 18 is different from the mode colour after line 18.

5 Demonstration of the Connection Utility Search

The aim of this experiment is to demonstrate the pruning performance of our move generator. Given the connection utility search extends the pattern search with our move generator, we can demonstrate the performance of our move generator by comparing implementations of these two search algorithms. The pattern search implementation will use a good static move generator called *Queen-bee* [1]. Thus, this experiment is comparing our move generator against a static move generator. In addition, an alpha-beta search with a Queen-bee move

generator is included as a control and as a benchmark. The search with the greatest move generator performance is the search which visits the least number of nodes (board positions) in the game tree, to completely solve an empty Hex board of a given size. Each search solves completely the 3x3 and 4x4 Hex board. The 4x4 Hex board is a practical limit in this experiment. However, additions such as a transposition table can push this limit to larger board sizes.

5.1 Results

Table 1 and Figure 7 show that our move generator maximizes pruning to a greater extent than a Queen-bee move generator in a pattern search. It also shows that a pattern search can perform better than an alpha-beta search given an appropriate move generator.

Search	Nodes Visited 3x3	Nodes Visited 4x4
Connection Utility Search	2223	3765784
Alpha Beta	3305	9871596
Pattern Search	9613	790811318

Table 1. The number of nodes visited for each search in the order of best-to-worst.

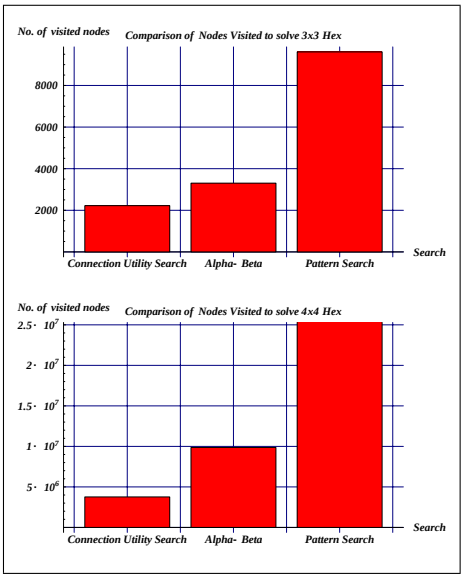

Fig. 7. A comparison of nodes visited by searches solving 3x3 and 4x4 Hex.

6 Conclusion

This paper presents a move generator which is based on connection utilities for solving Hex boards. Our experiments show, through the performance of our move generator, that connection utilities provide a good heuristic for ordering moves in a game of Hex. In recent developments, the connection utility search was modified so that threat patterns were reused via a specialized transposition table. With this development, our search completely solved the 5x5 Hex board in 165 nodes and the 6x6 Hex board in 77000 nodes. On a 3GHz Intel P4 machine, search times were about ten seconds and about ninety minutes, respectively. The same search without our move generator failed to solve the 6x6 Hex board after running for several days. We are currently improving threat pattern reuse and indexing in order to solve the larger sized boards.

References

1. van Rijswijck, J.: Computer Hex: Are Bees Better than Fruitflies? Master of science, University of Alberta (2000)
2. Hayward, R., Bjrnsson, Y., M.Johanson, Kan, M., Po, N., Rijswijck, J.V.: Advances in Computer Games: Solving 7x7 HEX: Virtual Connections and Game-State Reduction. Volume 263 of IFIP International Federation of Information Processing. Kluwer Achedemic Publishers, Boston (2003)
3. Hayward, R., Björnsson, Y., Johanson, M., Po, N., Rijswijck, J.v.: Solving 7x7 hex with domination, fill-in, and virtual connections. In: Theoretical Computer Science. Elsevier Science (2005)
4. Reinefeld, A.: A minimax algorithm faster than alpha-beta. In van den Herik, H., Herschberg, I., Uiterwijk, J., eds.: Advances in Computer Chess 7, University of Limburg (1994) 237–250
5. Müller, M.: Not like other games - why tree search in go is different. In: Fifth Joint Conference on Information Sciences. (2000) 974–977
6. Abramson, B.: Control strategies for two player games. ACM Computing Survey **Volume 21**(2) (1989) 137–161
7. van Rijswijck, J.: Search and Evaluation in Hex. Master of science, University of Alberta (2002)
8. Gardener, M.: The game of hex. In: The Scientific American Book of Mathematical Puzzles and Diversions. Simon and Schuster, New York (1959)
9. Anshelevich, V.V.: An automatic theorem proving approach to game programming. In: Proceedings of the Seventh National Conference of Artificial Intelligence, Menlo Park, California, AAAI Press (2000) 198–194
10. Anshelevich, V.V.: A hierarchical approach to computer hex. Artificial Intelligence **134** (2002) 101–120
11. Yang, J., Liao, S., Pawlak, M.: On a Decomposition Method for Finding Winning Strategies in Hex game. Technical, University of Manitoba (2001)
12. Rasmussen, R., Maire, F.: An extension of the h-search algorithm for artificial hex players. In: Australian Conference on Artificial Intelligence, Springer (2004) 646–657

7 Appendix

Algorithm 1 Connection_Utility_Search(*board*) **Returns** (Carrier, vector of Connection Utilities, Winner Colour)

```
1: utilities[1 : board.size] ← 0; carrier ← ∅
2: modeColour ← Black  {Start the search in Black mode}
3:
4: if board.isTerminal then
5:    modeColour ← board.winningPlayer
6:    for all i ∈ board.winnersStones do
7:       utilities[i] ← 1
8:    end for
9:    return(carrier, utilities, modeColour)
10: end if
11:
12: moveList ← aStaticMoveGenerator(board)
13:
14: while moveList ≠ ∅ do
15:    m ← popFirst(moveList)
16:
17:    board.playMove(m)
18:    (C, Util, winColour) ← Connection_Utility_Search(board)
19:    {The mode colour changes if winColour ≠ modeColour.}
20:    modeColour ← winColour
21:    board.undoMove(m)
22:
23:    if modeColour = board.turn then
24:       {m was a winning move.}
25:       carrier ← {m} ∪ C {carrier, is a weak threat pattern carrier.}
26:       return(carrier, Util, modeColour)
27:    else
28:       {m was not winning, try a remaining move.}
29:       carrier ← carrier ∪ C  {OR deduction rule}
30:       utilities ← utilities + Util  {vector operation}
31:       moveList ← moveList ∩ C  {the must-play region}
32:       moveList ← SortDescending(moveList, utilities)
33:    end if
34: end while
35: return(carrier, utilities, modeColour) {Successful OR deduction.}
```